Truly Scalable K-Truss and Max-Truss Algorithms for Community Detection in Graphs

Alessio Conte, Daniele de Sensi, Roberto Grossi, Andrea Marino, Luca Versari

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Truly Scalable K-Truss and Max-Truss Algorithms for Community Detection in Graphs

ALESSIO CONTE[1],DANIELE DE SENSI[1], ROBERTO GROSSI[1], ANDREA MARINO[2], AND LUCA VERSARI[3]

[1]University of Pisa, Pisa, Italy (e-mail: {conte,desensi,grossi}@ di.unipi.it)
[2]University of Florence, Florence, Italy, (e-mail: andrea.marino@ unifi.it)
[3]University of Pisa, Pisa, Italy and Google Research, Zurich, Switzerland (email: veluca@ google.com)

Corresponding author: Alessio Conte (e-mail: conte@ di.unipi.it).

This work was supported in part by the Italian Ministry of University and Research (MIUR) under PRIN Project n. 20174LF3T8 AHeAD (Efficient Algorithms for HArnessing Networked Data). A preliminary version of the algorithm presented here was a finalist in the MIT Graph Challenge 2018 and part of the contents of this paper appeared in [11].

ABSTRACT The notion of k-truss has been introduced a decade ago in social network analysis and security for community detection, as a form of cohesive subgraphs less stringent than a clique (set of pairwise linked nodes), and more selective than a k-core (induced subgraph with minimum degree k). A k-truss is an inclusion-maximal subgraph H in which each edge belongs to at least $k-2$ triangles inside H. The truss decomposition establishes, for each edge e, the maximum k for which e belongs to a k-truss. Analogously to the largest clique and to the maximum k-core, the strongest community for k-truss is the max-truss, which corresponds to the k-truss having the maximum k. Even though the computation of truss decomposition and of the max-truss takes polynomial time, on a large scale, it suffers from handling a potentially cubic number of wedges. In this paper, we provide a new algorithm FMT, which advances the state of the art on different sides: lower execution time, lower memory usage, and no need for expensive hardware. We compare FMT experimentally with the most recent state-of-the-art algorithms on a set of large real-world and synthetic networks with over a billion edges. The massive improvement allows FMT to compute the max-truss of networks of tens of billions of edges on a single standard server machine.

INDEX TERMS community detection, graph algorithms, in-memory computation, k-trusses, social network analysis, truss decomposition

I. INTRODUCTION

One of the most fundamental tasks in the analysis of real-world networks is that of community detection, which corresponds to identifying cohesive portions of a network according to some metrics. On one side suitable metrics should find communities that are meaningful and free from noise; on the other side, algorithms should be as fast as possible, since network sizes for many practically relevant problems are growing over the years. These two objectives are often in contrast with each other: simple metrics tend to be more efficient to compute but give lower quality results (e.g. core decomposition [26]) while others (e.g. based on cliques [12]) are rigorous but computationally heavy. Finding a good trade-off between performance and quality is a crucial problem, and great effort has been devoted to finding better metrics and/or better algorithms for those metrics [4], [10], [20], [33].

In this scenario a popular choice is the k-truss, a triangle-based cohesive subgraph introduced by Cohen [10] as one of the interesting patterns in social and communication graphs, such as those generated by phone calls, emails, and so on. It is defined as follows.

Consider an undirected graph $G = (V(G), E(G))$ with $n = |V(G)|$ nodes and $m = |E(G)|$ edges, where $N_G(v)$ represents the neighborhood of node v in G, and $\delta(v) = |N_G(v)|$ is v's degree [1]. We define a triangle in G as a set of three nodes u, v, z that are pairwise connected (i.e. they form a clique of size three). In that case, we say that edges

[1]We assume wlog that G does not contain isolated nodes, thus its size is $O(n+m) = O(m)$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

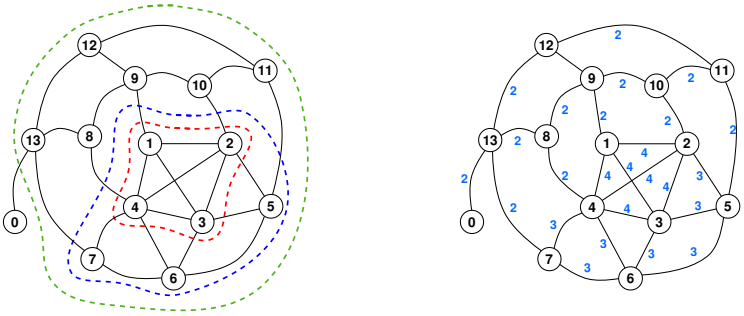


FIGURE 1. Left: a 3-core (green), 3-truss (blue) and 4-clique (red), respectively, of the graph G with trussness $t_G = 4$; note that the 4-clique is also a 4-truss (i.e. the max-truss here). Right: truss decomposition of the graph, where each edge is annotated with its trussness value: a k-truss can be obtained by singling out the edges with trustness value $\geq k$.

$\{u, v\}, \{v, z\}, \{z, u\}$ belong to the triangle. For each edge $e = \{u, v\}$ in G, its support $sup_G(e) = |N_G(u) \cap N_G(v)|$ is defined as the number of triangles to which e belongs.

Given an integer $k \geq 2$, the k-truss of G is the inclusion-maximal edge-induced subgraph H of G such that each edge e of H belongs to at least $k - 2$ triangles of H. Specifically, $H = (V(H), E(H))$ where $E(H) \subseteq E(G)$, $V(H) = \{x \in V(G) \mid \{x, y\} \in E(H)\}$, and $sup_H(e) \geq k - 2$ for every $e \in E(H)$.

Some examples of k-trusses are shown on the left of Figure 1, comparing them with k-cores (all the nodes have degree at least k in H) and k-cliques (all the nodes are pairwise connected in H). As it can be observed, k-trusses are more rigorous than k-cores but less stringent than cliques: it can be proved that a k-truss is a subgraph of a $(k-1)$-core, and that a k-clique is also a k-truss [2]. Furthermore, k-trusses can be computed in polynomial time, and may also be employed to quickly remove useless edges from a graph when looking for k-cliques [10].

The truss decomposition of G corresponds to assigning each edge its trussness value, i.e., the highest k for which the edge belongs to a k-truss (see Figure 1 on the right, where each edge is annotated with its trussness). Given the truss decomposition, it becomes easy to extract the k-truss for any k, and thus for the largest k, which is useful to identify the most important cohesive subgraphs. Furthermore, the k-truss of a graph is unique and can be obtained by performing peeling, i.e. recursively deleting edges that participate in less than $k - 2$ triangles [10].

The trussness t_G of the graph G is the maximum k such that there exists a k-truss in G, and the corresponding k-truss is called max-truss. It can be clearly obtained from the truss decomposition.

Over these years, the notion of k-trusses has spread in network analytics and is gaining momentum for other purposes other than security. For instance, the MIT/Amazon/IEEE GraphChallenge [23] organizes a benchmarking contest for triangle counting and k-truss discovery, with the best algorithms presented at the *IEEE High Performance Extreme Computing Conference*.

Several different algorithms have been proposed to compute k-trusses and truss decomposition [8], [10], [18], [19], [28], [30], [31]. In order to achieve good performance, they either rely on GPU computation or use significant amounts of memory, which is not always feasible or dramatically slow down their performance when dealing with large graphs. They deal with the bottleneck of the peeling process, after computing the triangles in G: when recursively deleting edges, some triangles disappear and the support of the corresponding edges must be updated. In turn, this either takes more time to recompute the supports from scratch or uses more space to store indexing data structures to find which edges have their support changed. We refer the reader to Section II for a discussion of the state of the art.

Our contribution.

We address the problem of finding the k-trusses in large networks from a new angle by introducing algorithm $FMT(G, \mathcal{M}, r)$, which takes in input a graph G, plus two parameters that interplay with the performance of the algorithm:

- $\mathcal{M}$ is the memory threshold for the computing platform
- r is an edge-pruning threshold to speed up the computation

If $\mathcal{M} = \infty$, FMT computes the truss decomposition of G in guaranteed $O(\alpha_G\, m)$ time using $O(m)$ space - specifically, approximately 32 bytes per edge (note that r is not employed in this case). Here α_G is the *arboricity* of G, which is the minimum number of forests into which $E(G)$ can be partitioned. It is related to the trussness, as $\alpha_G \geq \frac{t_G+1}{2}$, [3] and to $m = |E(G)|$, as $\alpha_G = O(\sqrt{m})$ (see [9]).

[2] Indeed a k-clique induces a k-truss: any two adjacent nodes in a clique of size k may form a triangle with any of the remaining $k-2$ nodes. Also, as each edge of a k-truss forms at least $k - 2$ triangles, its extremes must have degree at least $k-1$ in the k-truss, meaning that a k-truss is also a $k-1$-core. The implications in the other direction, however, are not true [10]. Moreover, for $k = 2$, the k-truss is trivially G as every edge participates in at least 0 triangles, while for $k = 3$ this is the set of edges that participates in at least one triangle.

[3] Since the degree of each node u in a k-truss is $\delta(u) \geq k + 1$, we have that a max-truss has at least $n'(t_G + 1)/2$ edges, where n' is the number of its nodes and $k = t_G$. As each forest covers at most $n' - 1$ edges in the max-truss, it yields $\alpha_G \geq \frac{t_G+1}{2}$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

On top of these theoretical guarantees, we carefully study the operations required by the algorithm: We engineer solutions that simultaneously use very few bits per edge and allow the usage of low-level instructions, while maintaining a structure that allows the algorithm to be easily parallelized. More details are given in Section IV (*algorithm engineering*).

As a result, our execution time on real-world networks compares favorably with the most recent state-of-the-art approaches for truss decomposition, coming from the GraphChallenge [23] and other papers on large real-world and synthetic graphs, as detailed in Section VI, with only one solution (KM17) having slightly less space requirement per edge. We show that our algorithm outperforms all similar algorithms on large networks (i.e., with millions of edges or more) by up to orders of magnitude. Moreover, using the USD cost per hour of equivalent infrastructure in Google Compute Engine (https://cloud.google.com/compute/), our algorithm results in significant financial savings, such as processing the largest amount of edges per USD cent, except for one dataset, relative to other algorithms.

If $\mathcal{M} \neq \infty$, FMT finds the max-truss in G. We believe that this is a further important contribution of this paper for the following reasons.

- The problem of finding the max-truss in G or its trussness t_G is quite natural. It has a similar flavor to that of finding the largest clique and the maximum k-core, or just their size, when computing all cliques or the k-core decomposition: these structures give important information for network analysis and as such their efficient computation is widely studied [14], [15], [29], [32] (in particular, the maximum k for a k-core exists is also known as the *degeneracy* and is used as a sparsity measure [15], [16]).
- We comment on real-world datasets in Section III, where the max-truss provides often meaningful communities. This has a good foundation in Section IV, where we link for the first time the trussness t_G to the densest subgraphs of G in terms of the number of triangles per edge (see Theorem 1, which is an extension of the Nash-Williams's theorem [1] to the trussness).

It makes sense to investigate the problem of quickly finding the max-truss and the trussness of G, since all existing approaches require to find a whole truss decomposition to do so.[4] When $\mathcal{M} \neq \infty$ or in general the memory is limited with respect to the size of the graph, such a truss decomposition cannot be found. In this scenario, $FMT(G, \mathcal{M}, r)$ avoids a whole truss decomposition and in particular, its novelty relies on the following new ideas.

- We introduce the notion of approximated trussness, and a suitable approximation algorithm (which depends on $\mathcal{M}$ and r) as a core routine to focus on the most promising parts of G.
- We use the approximation algorithm to design an algorithm to compute *exactly* the max-truss, using the following two parts.
  - The first part uses $\mathcal{M}$ and r to quickly compute lower and upper bounds on the trussness t_G while shrinking G by removing low trussness edges, which are likely not in the max-truss. A small residual graph is obtained in this way.
  - The second part decomposes the residual graph and identifies its max-truss. Using a suitable mechanism to check if the latter is not the max-truss of G, the algorithm is restarted in that case using the (possibly improved) lower bound found.

Interestingly, we also give a conditional lower bound that the computing time for the approximated trussness cannot be significantly smaller than that for the exact trussness t_G in the worst case.

The worst-case cost of FMT is still $O(\alpha_G\, m)$ time using $O(m)$ space. However, space is reduced approximately from 32 bytes per edge to $\max(8m, 32\min(\mathcal{M}, m))$ bytes per edge: setting $\mathcal{M} = m/4$, it gives 8 bytes per edge and allows us to analyze very large networks. Moreover, in Section IV we show that improving over $O(\alpha_G\, m)$ time is hard, as we give some conditional lower bounds for computing truss decomposition and max-truss that match our upper bounds. In particular, a faster worst-case time would improve Boolean matrix multiplication and other well-known problems.

Experiments in Section VI exhibit the benefits of our new approach. FMT sensibly reduces the time and space required with respect to the truss decomposition, still computing exactly the max-truss. In this way, we hope to shed further light on this popular community measure, both in its complexity and its relation to triangle density. At the same time, based on this knowledge, we provide scalable and efficient tools for computing the truss decomposition and the max-truss, which outperform known approaches and can process graphs with billions of edges in reasonable time and space. We also show in Section VI-B that, existing algorithms become less cost-effective when executed on graphs with larger k-trusses. Indeed, compared to our algorithm, existing algorithms perform some additional re-computation, which limits the achievable performance. Moreover, they rely on the use of GPUs, which increase the hardware cost compared to our CPU-based solution.

A preliminary version of the algorithm presented here was a finalist in the MIT Graph Challenge (graphchallenge.mit.edu/champions) and part of the contents of this paper appeared in [11].

II. RELATED WORK

Several notions of communities have been introduced in literature trying to find the right granularity and the trade-off between a huge number of highly clustered communities and a smaller number of poorly connected communities. The usefulness and the right trade-off depends on the context of applications. Trying to limit the number of communities found,

[4]Finding the max-truss is not to be confused with the simpler problem of finding "the k-truss" for a given k (see, e.g., [21]), as the trussness of the graph is not known a priori and—as we will see—is hard to compute.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

i.e. bounded by a polynomial, k-cores and k-truss are the most used approaches, even though other approaches [20], [33] for community discovery have been proposed.

In this paper, we focus on the well-known truss decomposition. The seminal paper about truss decomposition is that by Cohen [10], who introduced the concept of k-truss, motivating it as an effective community indicator, with applications to networks where community structure is relevant, such as social networks. The paper also presents a simple algorithm for the truss decomposition, whose structure is surprisingly effective: most known algorithms, that either use matrix multiplication or combinatorial approaches, are still essentially based on this structure. One of the first papers addressing the truss decomposition in massive networks [31] engineered Cohen's algorithm to improve performance on large graphs. Smith et al. proposed a parallel algorithm for truss decomposition in a shared-memory setting [28] (SL+17 in Section VI). The sequential algorithm is based on the algorithm by Wang et al. [31] (i.e., the structure of Cohen's algorithm), and keeps track of the support of edges using buckets, which are used to parallelize the algorithm. The parallel version scales up to 28 threads with a good speedup. SL+17 was one of the finalists of the 2017 GraphChallenge [23]. Another shared memory parallel algorithm which also improves upon Wang et al. [31] has been proposed by Kabir et al. [19] (KM17 in Section VI). However, since there is no direct comparison between KM17 and SL+17, we have considered both KM17 and SL+17 as our direct competitor. Further algorithms have been recently proposed by Wu et al. [36] and Davis et al. [13] (resp. WG+18 and D18 in Section VI). The former is serial and it is designed to work in Java with the WebGraph framework [7], the latter is parallel and in C.

The distributed algorithm by Pearce [25] has been champion of the 2018 GraphChallenge for k-truss decomposition (PS18 in Section VI). At the same GraphChallenge, the GPU algorithm by Date et al. [22] (MD+18 in Section VI, has been finalist. Further work has been done by Huang et al. [18] to consider the dynamic version of the problem which is: given a graph subject to edge deletion, efficiently answer the query "find the k-truss involving a node v". A distributed algorithm for truss decomposition has been proposed by Chen et al. [8]. Green et al. [17] consider finding max-truss, using a dynamic graph formulation on GPU. The times reported in the paper are much higher than the ones we have presented in Section VI, for instance for `as-Skitter`, even if our machine is sensibly slower.

We remark that truss decomposition and k-truss computation often benefits of the advances coming from triangle counting. To this aim, Wolf et al. [35] propose a high-performance parallel algorithm that uses linear algebraic matrix operations implemented with KokkosKernels, showing both a CPU and GPU implementation. Parallelization is achieved in another work by Pearce [24] through delegate nodes, which are used to partition the input graph. Bridging triangle counting and k-trusses, Bisson et al. [5] use parallel matrix multiplication, implemented on GPU, for counting triangles and computing the k-truss for a given k. Eventually, Voegele et al. [30] also consider computing the k-truss for a given k, using edge list intersection rather than matrix multiplication. The algorithm is based on the Cohen [10] algorithm, and gains efficiency by focusing first on edges incident to lower degree nodes and truncating list intersection whenever enough elements are in the intersection. It also uses the property that a k-truss is necessarily a $k-1$-core.

III. DATA ANALYSIS

In the literature k-trusses are considered one of the powerful tools for community analysis in networks, because they permit to focus on interesting portions of the graph at hand. For instance, the truss decomposition of a network G has been employed in [18] to query the communities for any given node v_q of G and a positive integer k. It consists in finding the largest connected subgraph of G having trussness k and containing v_q: the idea is to suitably modify a graph traversal beginning from v_q, and using a queue to store and explore the edges that have trussness $\geq k$. In the truss decomposition shown on the right in Figure 1, if $v_q = 5$ and $k = 3$, the corresponding community is the subgraph induced by nodes $1, 2, 3, 4, 5, 6, 7$. We refer the reader to Algorithm 2 in [18] for further details. In this scenario, considering the max-trusses specializes the query for the more cohesive subgraphs (i.e. for the largest feasible k), when memory is limited.

Depending on the network's topology, the communities found, as mentioned above, can be further analyzed and refined by inspection. In order to illustrate this task, we consider four datasets taken from LAW (http://law.di.unimi.it/). Their size and statistics are shown in Table 1 (we refer the reader to Section VI for performance analysis of our approach).

Web snapshot: `gsh-2015-host`

This is the host graph of the graph `gsh-2015`, which is a large snapshot of the web (988M nodes and 33G edges) taken in 2015 by BUbiNG [6] starting from the site `http://europa.eu/` without any domain restriction. In the host graph, pages with the same domain name (host) are collapsed; the maximum number of pages per host was set to 100, to find a large number of hosts, and the resulting host graph is composed of 68M nodes and 1.5G edges.

In this graph, the max-truss is composed of a unique connected component of 9960 nodes. Interestingly, the great majority of the nodes are of the form `www.XYZ.de`, for some word `XYZ` of 4 letters, like for example `www.aafd.de` or `www.sfnc.de`. The content of the pages inside the component is almost duplicate, in the sense that they differ just by replacing `XYZ` in the text. The links to other domains of the group are hidden in the pages' source. All these domains are related to the website `www.verleihcenter.eu` and we have found, by private communication, that they have been registered on purpose by the owner, who seems to be a domain collector. We observe that this k-truss helped to identify

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

a large set of near-duplicates, which is a desideratum when crawling the web. Interestingly, all these domains do not cluster into just one maximal clique, but rather in many maximal cliques (more than 350000 cliques with 100 nodes or more), meaning that the strict notion of clique may not always allow easy identification of densely interconnected substructures, while the looser notion of k-truss may in some cases be more meaningful despite being faster to compute. Decreasing k, for instance setting $k = 4000$, we find a component whose nodes are subdomains of the website `iloan24.com`, having as URLs `paydayloansXYZ.iloan24.com` for some string `XYZ`. We conjecture these have been created to increase the visibility of `iloan24.com`.

Movie actors: `hollywood-2011`

In this graph extracted from the Internet Movie Database (IMDb) vertices are actors, and two actors are joined by an edge whenever they appeared in a movie together. The max-truss corresponds to a community of 1298 nodes, which is a clique of the actors starring in the movie "Around the World in Eighty Days (1956)". The movie, based on the homonym novel by Jules Verne, follows Victorian Englishman Phileas Fogg (David Niven) in several locations around the globe, and thus stars a significant amount of background actors. Notably, it seems that many of these were registered in the IMDb database, resulting in this large clique. Decreasing k to $k = 1000$ we obtain a connected k-truss of size 3529, composed by 3 cliques of similar size: the above clique, one clique of actors active in the 80s (mostly Spanish), and one containing Hollywood superstars (e.g. Quentin Tarantino, Uma Thurman, and Jessica Alba). Frank Sinatra, Shirley MacLaine connect the first two cliques, while Martin Scorsese is in the second and the third one.

Wikipedia snapshot: `enwiki-2013`

This graph represents a snapshot of the English-language Wikipedia as of February 2013. The max-truss corresponds to a unique connected component with 335 nodes. This is mostly composed of pages referring to years and dates, which occur often in common pages. By decreasing k, setting for instance $k = 40$, we obtain four connected components of size respectively 226, 308, 267, and 570. The last component is still related to dates and years frequently co-occurring, but, notably, the other components contain Wikipedia pages about different sports: the first is about NHL (National Hockey League), the second relates to tennis (Serena Williams, Rafael Nadal, and Roger Federer are nodes of this community), the third relates to English football (teams, coaches, stadiums, history of English football). Once again, it is worth observing the presence, in each component, of many maximal cliques which are difficult to aggregate into a single community, while connected k-trusses seems to isolate and aggregate the data quite accurately.

Algorithm 1: Our algorithm FMT

Input : graph $G = (V(G), E(G))$, memory threshold $\mathcal{M} > 0$, approximation factor $r \geq 1$
Output: trussness t_G and max-truss in G.

$u \leftarrow 0$, $\ell \leftarrow 0$
while $|E(G)| > \mathcal{M}$ **do** // Part I
  $\mathcal{T}_G \leftarrow \frac{1}{3}\sum_{e \in E(G)} sup_G[e]$
  $\ell \leftarrow \max(\ell, minsup(G), \frac{\mathcal{T}_G}{|E(G)|})$
  $u \leftarrow \max(u, minsup(G), r\frac{\mathcal{T}_G}{|E(G)|})$
  $Q \leftarrow \{e \in E(G) : sup_G[e] \leq u\}$ // *edges to delete*
  $G \leftarrow G \setminus Q$ // *remove Q's edges from $E(G)$*
$t \leftarrow u$
while $E(G) \neq \emptyset$ **do** // Part II
  $Q \leftarrow \{e \in E(G) : sup_G[e] \leq t\}$ // *edges to delete*
  $G \leftarrow G \setminus Q$ // *remove Q's edges from $E(G)$, update $\mathcal{T}_G$*
  if $t > u$ **then** **foreach** $e \in Q$ **do** $truss[e] \leftarrow t + 2$
  $t \leftarrow \max(t, minsup(G))$
if $t > u$ **then** // *values $> u + 2$ in $truss[]$ are exact*
  return $truss[]$, $t_G = t + 2$
else // $\ell + 2 \leq t_G \leq r(\ell + 2)$
  $r \leftarrow 1$, $u \leftarrow \ell$, $G \leftarrow$ input graph // *reset parameters*
  goto step 2

Bibliographic database: `dblp-2011`

The graph is a 2011 snapshot of the scientific bibliography service DBLP, from which an undirected collaboration network can be extracted: each vertex represents a scientist and two vertices are connected if they co-authored an article. The max-truss is composed of the 119 authors of the paper Length Sensing and Control in the Virgo Gravitational Wave Interferometer. IEEE Trans. Instrumentation and Measurement 55(6): 1985-1995 (2006). As this is a clique and it is the max-truss, it means that this is the maximum clique in the graph, otherwise, a larger clique would have implied the presence of a larger k-truss. This suggests that k-trusses can be an effective way to spot large (and sometimes maximum) cliques.

IV. OUR ALGORITHM

As described in Section I, $FMT(G, \mathcal{M}, r)$ consists of two parts, and its pseudocode is shown in Algorithm 1.

Part I: approximation

The goal of this part is to prune G as quickly as possible to less than $\mathcal{M}$ edges so that it fits in main memory, trying to only remove edges with *low* trussness, and in particular

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

without removing any from the max-truss [5].

A usual peeling algorithm requires random access to the graph and supporting data structures for quick updates. What we do instead is computing all supports at once by listing the triangles of G, an operation which is heavy but can be done efficiently with a much more compact representation of the graph (around 8 bytes per edge, as detailed later). Then, we remove edges with "low support" and repeat. To achieve a performance benefit, we want to perform this step as few times as possible, and so remove each time a large number of edges.

Let $\mathcal{T}_G$ be the number of triangles in G. From Section V, we exploit the inequality that $t_G \geq \mathcal{T}_G/|E(G)| + 2$ (implied by Theorem 1 there): we can remove all edges with support lower than $\mathcal{T}_G/|E(G)|$ without impacting the max-truss. At the same time, $\ell \leftarrow \mathcal{T}_G/|E(G)|$ gives us a lower bound on the trussness t_G (as well as the minimum support in the current graph). As soon as we remove enough edges ($|E(G)| \leq \mathcal{M}$), we start Part II on the residual graph. To obtain a greater speedup, we further increase the support threshold to $u \leftarrow r\,\mathcal{T}_G/|E(G)|$; the larger r is, the more edges we remove at each step. The drawback is that u might exceed $t_G - 2$ before the end of the part, and cause the removal of edges in the max-truss: if so, this will cause more operations in Part II. In practice, suitable choices of $\mathcal{M}$ and r (see discussion in Section VI-C) can avoid this, and prevent these additional operations.

Part II: max-truss refinement

For the sake of discussion, let G' denote the residual graph G obtained from Part I (so as to distinguish it from the input graph G). We perform an optimized peeling (using around 32 bytes per edge) to compute the truss decomposition of G', albeit ignoring the supports $\leq u + 2$ (*if* in Line 12). This further optimizes the algorithm as we can immediately peel off all edges with support u or lower. Since all edges in a $(u+3)$-truss have support at least $u+1$, edges with trussness $u+3$ or higher in G are still in G' and have the same trussness, thus the decomposition will be correct for all values of trussness larger than $u+2$, which gives us a partial truss decomposition and the max-truss. If no such value is found, it means Part I deleted edges from the max-truss. We thus restart the algorithm, but we can ignore all supports smaller than ℓ, which is a lower bound and a r-approximation of the trussness, meaning that the procedure will still be faster than a full decomposition. In this case, we also set r to 1, which makes sure no edge of the max-truss will be accidentally removed (by Theorem 1).

As a special case, setting $\mathcal{M} = \infty$ (and any value of r), it gives an algorithm for computing the truss decomposition, as, it essentially skips Part I and starts Part II with $G' = G$ and $u = 0$. We denote this as $FMT - dec = FMT(\infty, \cdot)$. As we show in Section VI, it compares favorably with existing algorithms for truss decomposition, thanks to the optimized and parallel friendly structure of Part II.

Algorithm engineering

We briefly detail the key operations of FMT that contribute to its performance.

In both Part I and Part II, we employ a careful implementation of list intersection, that makes use of SIMD instructions (specifically, SSE4.1 instructions) when the two lists to intersect have similar length. When one list is significantly longer (we set this to be by a factor 2 or more, which gave the most consistent benefits), we employ a binary-search based approach where the next common element between the lists is found by at most 2 binary searches.

In Part I, the algorithm writes a first file containing for each node, its degree followed by the list of its neighbors with larger id (assume the nodes labeled as integers $1, \ldots, n$), a second file containing pointers to the start of each adjacency list in the previous file, and a third one containing the support computed for each edge, associated with said edge at no extra cost by storing them in the same order as in the first file. This compact representation takes roughly 8 bytes per edge [6], but is enough for listing G's triangles [7].

The files are dynamically mapped to memory using the `mmap()` function so that paging is left to the OS low-level routines. After a set Q of edges is removed, the files are suitably updated before the next step of support computation. Note that the files are fully loaded in memory whenever few enough edges are left, however, this does not immediately trigger the next part, as Part II involves a larger number of bytes per edge.

Once Part II starts, the residual graph is loaded in memory as follows. We store the concatenated adjacency lists as above, including all neighbors (not just the ones with larger id). We store a pointer to the start of each node's adjacency list, and for each edge, a pointer to the lists of its endpoints. We sort the edges $e \in E(G)$ in increasing order of $sup_G(e)$, storing the edges in buckets corresponding to their support. These structures can be built in $O(m)$ time, and take around 32 bytes per edge, but allow identifying an edge of minimum support, its endpoints, and changing the bucket of an edge (when its support is updated), in $O(1)$ time.

As long as the buckets are nonempty, we remove an edge $e = \{u, v\}$ of minimum support from its bucket, and decrease by 1 the support of the edges forming a triangle with e, i.e., those in $\{\{u, z\}, \{v, z\} \mid z \in N_G(u) \cap N_G(v)\}$. As previously observed, this can be done by looking at the endpoint of e with the smallest degree, in $O(\min\{\delta(u), \delta(v)\})$ time. This structure allows easy and scalable parallelization, as we can remove multiple edges from the lowest support bucket at

[5] We reasonably assume that $\mathcal{M}$ is large enough to include the max-truss.

[6] Assuming we can represent nodes with unsigned 4-bytes integers, which is the case for graphs with up to 4 billion nodes; otherwise, memory is doubled. 12 bytes per node are also used, however, this is usually not significant as n is smaller than m by at least a factor 10.

[7] Indeed, any triangle i, j, k (with $i < j < k$) is found by intersecting the larger neighbors of i with those of j, as both contain k.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

once (as they all need to be removed by the algorithm) and intersect the neighborhoods of their endpoints, the heaviest task, in parallel.

Analysis.

Since each edge is removed once, the total cost of Part II is $O(m + \sum_{\{u,v\}\in E(G)} \min\{\delta(u), \delta(v)\}) = O(m\,\alpha_G)$ time and $O(m)$ space, since $\sum_{\{u,v\}\in E(G)} \min\{\delta(u), \delta(v)\} \leq 2m\,\alpha_G$, see [9]. By the same logic, the cost of Part I is also $O(m\,\alpha_G)$ times the number of support computation steps.

As previously noted, we may set $\mathcal{M} = \infty$ and obtain an algorithm for complete truss decomposition by executing only Part II (called $FMT-dec$), we get the following result.

Lemma 1. *Given a graph G with m edges and arboricity α_G, FMT-dec computes its trussness in $O(m\,\alpha_G)$ time and $O(m)$ space.*

We remark that FMT-dec can count and list all the triangles within the same complexity as above.

As for the total memory usage, we can bound it to approximately $\max(8m, 32\min(\mathcal{M}, m))$ bytes (triggering external memory usage if $8\,m$ exceeds available RAM), and simply $32\,m$ bytes for FMT-dec. This is confirmed in practice by our experiments, where this bound is never exceeded by more than 10%.

V. THEORETICAL BASIS

If $\mathcal{T}_G$ is the number of triangles in G, we can see that $t_G \geq \mathcal{T}_G/m + 2$ as an instance of the following extension of Nash-Williams' result [1] to trussness.

Theorem 1. *Given an undirected graph G with trussness t_G, let $\mathcal{T}_S$ be the number of triangles and m_S be the number of edges in any subgraph S of G. Then $\max_{S\subseteq G} \frac{\mathcal{T}_S}{m_S} \leq t_G - 2 \leq 3\max_{S\subseteq G} \frac{\mathcal{T}_S}{m_S}$*

Proof. We first prove that there exists a subgraph S of G such that $3\frac{\mathcal{T}_S}{m_S} \geq t_G - 2$, thus proving the upper bound on the trussness. Indeed, let S be a t_G-truss of G. Since $3\mathcal{T}_S = \sum_{e\in E(S)} sup_S(e)$, and since each edge has a support in S of at least $t_G - 2$, it follows that $3\mathcal{T}_S \geq m_S(t_G - 2)$, i.e. $3\frac{\mathcal{T}_S}{m_S} \geq t_G - 2$.

For the lower bound, we observe that $t_S \leq t_G$ for any subgraph S of G. It follows that it suffices to prove the inequality for $S = G$, namely, $\mathcal{T}_G \leq m\,(t_G - 2)$, as it also implies $\mathcal{T}_S \leq m_S\,(t_S - 2) \leq m_S\,(t_G - 2)$ when applied to S.

We will prove this by induction on the number of edges m. The base case is trivial, as a graph with no edges has no triangles. For the inductive step, notice that the graph must have an edge e with support at most $t_G - 2$, as otherwise G would have a $(t_G + 1)$-truss, contradicting the definition of trussness. Thus, if we consider the graph G' obtained by removing e from G, we have $\mathcal{T}_G \leq \mathcal{T}_{G'} + t_G - 2 \leq m_{G'}(t_{G'} - 2) + t_G - 2 \leq (m-1)(t_G - 2) + t_G - 2 = m\,(t_G - 2)$, concluding our proof. □

We provide computational lower bounds for computing the trussness of a graph. As the trussness can be obtained by computing the truss decomposition, these bounds hold also for the computation of the truss decomposition, as stated next.

Theorem 2. *Given any undirected graph G with m edges, arboricity α_G and trussness t_G, triangle counting/listing and graph trussness cannot be computed by combinatorial algorithms in either $o(m\,\alpha_G \log^{O(1)} m)$ time or $O(mt_G)$ time in the worst case, unless Boolean matrix multiplication is truly subcubic [34].*

The proof of Theorem 2 follows from the fact that trussness (and computing the max-truss and the truss decomposition) is intimately related to triangle-free graphs.

Fact 1. *G is triangle-free if and only if its trussness t_G is 2.*

Note that triangle counting has better time complexity than triangle listing when matrix multiplication is employed [3]. When we refer to a "combinatorial" approach, we mean that it does not use matrix multiplication. We can use in this way a well known conditional hardness result.

Theorem 3. (Theorem 1.3 from [34]) The following all have truly subcubic "combinatorial" algorithms, or none of them do:

- Boolean matrix multiplication (BMM).
- Detecting if a graph has a triangle.
- Listing up to $n^{3-\delta}$ triangles in a graph for constant $\delta > 0$.
- Verifying the correctness of a matrix product over the Boolean semiring.

Improving the worst-case cost of computing the trussness to significantly less than $O(m\,\alpha_G)$ time using combinatorial algorithms is quite hard because of Theorem 3 and $m\,\alpha_G = \Theta(n^3)$ in the worst case. Since trussness t_G is also a parameter for complexity analysis, one could hope to get $O(m(t_G + 1))$ time instead of $O(m\,\alpha_G)$ time. Not even this is possible because of Fact 1, since triangle free graphs have $t_G = 2 = O(1)$, meaning we could recognize if G is triangle-free in linear time. In summary, we obtain the result in Theorem 2.

VI. EXPERIMENTS

This section is devoted to showing the performance of FMT, compared to the fastest known algorithms.

State-of-the-art algorithms

We will compare our algorithm FMT-dec for finding the truss decomposition with the following ones, which are the most recent state of the art algorithms for k-truss computation.

1) SL+17: a parallel shared-memory algorithm proposed in [28], finalist in the 2017 GraphChallenge [23]. The code has been downloaded from https://github.com/KarypisLab/K-Truss.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

NETWORK	TYPE	NODES	EDGES	t_G
flickrEdges	img	105 938	2 316 948	574
Amazon0505	prod	410 236	2 439 436	11
dblp-2011	coll	986 324	3 353 618	119
as-Skitter	as	1 696 415	11 095 298	68
cit-Patents	cit	6 009 555	16 518 947	36
enwiki-2013	soc	4 206 785	91 939 728	53
hollywood-2009	coll	1 139 905	56 375 711	2 209
hollywood-2011	coll	2 180 759	114 492 816	1 298
g500-sc23-ef16	rand	6 323 640	129 250 705	625
g500-sc25-ef16	rand	17 043 781	523 467 448	996
arabic-2005	web	22 744 080	553 903 073	3 248
it-2004	web	41 291 594	1 027 474 947	3 222
twitter-2010	soc	41 652 230	1 202 513 046	1 998
gsh-2015-host	web	68 660 142	1 502 666 069	9 923
com-Friendster	soc	65 608 366	1 806 067 135	129
gsh-2015	web	$988{\cdot}10^6$	$25.7{\cdot}10^9$	5 204
eu-2015	web	$1.1{\cdot}10^9$	$80.5{\cdot}10^9$	13 049

TABLE 1. Graphs considered in our experiments.

2) KM17: proposed in [19] and won a student innovation award in the same challenge. The code has been downloaded from https://github.com/humayunk1/PKT.
3) WG+18: serial algorithm in [36], designed to process large graphs on consumer-grade hardware using the WebGraph framework [7]. Code kindly provided by the authors [8].
4) D18: the best performing algorithm proposed in [13], that is the "highly optimized", parallel implementation in C of the *all k-truss* algorithm. Code kindly provided by the author.
5) PS18 is the champion of the 2018 GraphChallenge for k-truss decomposition, running on distributed settings [25].
6) MD+18 is a finalist of the 2018 GraphChallenge for k-truss decomposition, specifically designed for GPU platforms [22].
7) AA+19 is a winner of the Student Innovation Awards in the 2019 GraphChallenge [2]

To the best of our knowledge, all other known methods for k-truss decomposition in literature are directly improved or outperformed by at least one of these (see Section II for discussion).

Moreover, we will consider our algorithms for finding the max-truss. In particular, we will use FMT setting $r = 4$ and $\mathcal{M} = m/10$, where m is the number of edges in the input graph G. In the following, we will refer to this method as FMT-max. We compare these with AA+19 [2], which also proposes an algorithm for max-truss computation.

For more details about the reason behind this choice, we refer to Section VI-C, where we have shown the behaviour of FMT when varying its input parameters.

Direct Comparison.

As other algorithms, namely SL+17, KM17, WG+18, and D18, are designed for platforms similar to ours, we will compare the memory usage and the execution time of our algorithms with respect to them, running all these algorithms on our platform. The performance measures we considered are the ones used also in the GraphChallenge 2017 and 2018.

Indirect Comparison.

The latter algorithms are either GPU-based (MD+18 and AA+19), or designed for a cluster of multicores (PS18). This makes a direct comparison challenging (furthermore, the software is not available). To obtain a meaningful comparison, we devised a cost-based approach that is detailed in Section VI-B, which takes into account the cost of the machines used and the time needed by that machines to conclude an experiment, using the results and dataset reported in the corresponding papers [2], [22], [25].

Replicability: Source Code and Computing Platform.

The computing platform is a machine provided by the University of Pisa with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, 24 virtual cores, 128 Gb RAM, running Ubuntu Linux version 4.4.0-22-generic. The program is written in `C++11`, compiled with `gcc-8.1.0`, using the `-O3` optimization flag. [9] `OpenMP` has been used to implement the parallel version of our code, and we used `AVX2` for instruction-level data parallelism.

Dataset

Our dataset, shown in Table 1, includes networks for which community detection is relevant, including collaboration, autonomous systems, social, and web networks, taken from LAW (law.di.unimi.it/), Graph500 (graph500.org), and SNAP (snap.stanford.edu/). We report in the table, for each network, its abbreviation, type, number of nodes and edges, and the trussness.

Structure of the Experiments.

The experiments are organized as follows. In Section VI-A, we firstly compare FMT with its direct competitors using single cores and small networks. We then compare these methods in a parallel setting when dealing with large networks and we analyze their scalability when varying the number of cores. In Section VI-B, we perform our indirect comparison.

A. DIRECT COMPARISON

In this section, we analyze the performance of our algorithms with respect to SL+17, KM17, WG+18, and D18. For a fair comparison, as all the competitors are designed to perform truss decomposition, we will compare them with our FMT-dec, which also returns the truss decomposition. Moreover, we will show the performance of FMT-max in order to find just the max-truss. We first evaluate their performance using a single core and then exploiting their parallelism.

[8] The paper proposes a parallel algorithm too, but (as confirmed by the authors) the serial one is consistently faster even in parallel environments.

[9] The source code of our algorithm can be found at github.com/google-research/google-research/tree/master/truss_decomposition

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

NET	MAX-TRUSS	DECOMPOSITION				
	FMT-max	FMT-dec	SL+17	KM17	D18	WG+18
flickrEdges	0.60	**0.61**	12.79	11.6	6 325.43	76.47
Amazon0505	0.96	2.24	2.84	**1.94**	16.99	9.43
dblp-2011	1.08	2.67	3.28	**1.77**	38.8	11.86
as-Skitter	3.43	**12.34**	16.35	50.96	3 146.23	99.19
cit-Patents	6.37	16.7	13.42	**13.12**	46.02	41.59
enwiki-2013	58.60	**250.98**	596.62	1 742.7	*oot*	2 199
hollywood-2009	158.09	**313.07**	768.13	612.5	*oot*	4 815
hollywood-2011	303.78	**765.18**	1 764.34	1 359	*oot*	12 576
arabic-2005	837.67	**1 284.40**	*oom*	14 426.15	*oot*	*oot*

TABLE 2. Execution time comparison (sec.) in the sequential setting for smaller graphs. *oot*: required more than 30 000 seconds.

1) Performance Evaluation on a Single Core

In this section, we compare the time needed by FMT-dec with respect to the one used by SL+17, KM17, WG+18, and D18, using just one core in order to test the effectiveness of the underlying algorithms independently from their parallel aspects. For this reason, we ran the competitors on a restricted number of networks, which are the smaller networks in our dataset. We report in the right part of Table 2 (decomposition) our results. This table shows that KM17 is faster in 3 small graphs, all of them having less than 17 millions of edges. On the other hand, SL+17, D18 and WG+18 are never faster than the others. SL+17 clearly outperforms KM17 on enwiki-2013, but it runs out of memory when dealing with bigger ones, like arabic-2005. In this scenario, FMT-dec outperforms the competitors on the four biggest graphs, being the fastest for all the graphs having more than 20 million edges. For these graphs, the time saved by FMT-dec goes from the order of minutes to the order of hours (when the competitors can end their experiments). The improvement is more visible on the largest graph in the table, arabic-2005: FMT-dec spent less than 22 minutes to process it, while the only competitor able to conclude the experiment was KM17, which used more than 4 hours.

In the left part of Table 2 (max-truss), we report the time needed by FMT-max to compute the max-truss. It is worth observing that on the smaller graphs, such as flickrEdges, the difference between FMT-dec and FMT-max is negligible as the time is not dominated by the algorithm but its side aspects, e.g. input and output. On the other hand, the effectiveness of FMT-max is more evident on bigger graphs, as it allows us to save a conspicuous amount of time, namely half of the total time, without requiring to compute the whole truss decomposition. Its effectiveness will be even more evident on the bigger graphs we will consider in the remaining part of this section.

2) Dealing with Large Scale Networks using Multiple Cores

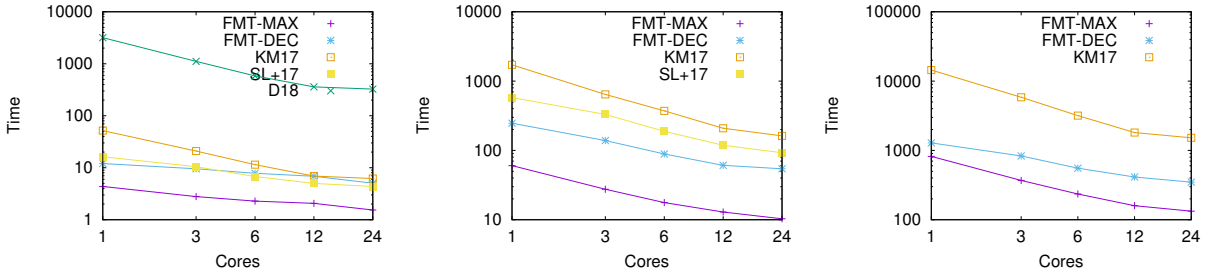
In the right part of Table 3 (having headline decomposition), we report the experimental results to compute the truss decomposition for all the networks in our dataset using 24 parallel threads. Looking at the results, we can see that FMT-dec is always faster than the competitors on bigger graphs. In the case of gsh-2015-host for instance, FMT is more than eight times faster than KM17, which was the only competitor able to conclude the computation on this graph. Concerning the other methods, it is worth mentioning that WG+18 is not parallel and, hence, it uses just one core. Nonetheless, we also tried to run WG+18 on bigger graphs, but, due to the lack of parallelism, it ran out of the allotted 30 000 seconds. Also, D18 ran out of time for bigger graphs. On the other hand, SL+17 appears to be fast on some medium-sized graphs, like on g500-sc23-ef16 and g500-sc25-ef16, but its higher memory usage prohibits its application on larger networks on our workstation. Indeed, it should be noted that the original paper [28] was able to run SL+17 on large networks due to the larger main memory available (384GB against the 128GB of our machine). Among the competitors of FMT-dec, KM17 seems to be faster than the others on the majority of the graphs, even though there seems to be variability in the results, as for some graphs it is heavily outperformed by SL+17. Indeed, KM17 is the only competitor of FMT-dec able to process some of the largest graphs in our dataset. This is due to its optimized usage of the main memory, which is shown for the sake of completeness in Table 4 for the biggest graphs. Note that in this table, WG+18 and D18 do not appear as they ran out of time (see Table 3). Looking at the table, we note that KM17 is indeed the one using less memory, namely much less than SL+17, and slightly less than our approach.

Finally, we discuss the execution time of FMT-max to compute just the max-truss instead of the whole truss decomposition. The second column of Table 3 shows that FMT-max allows us to further reduce the time usage with respect to the competitors which have to perform the whole truss decomposition to discover the max-truss. In particular, FMT-max is at least one order of magnitude faster than KM17, as also of the other methods. The reason for this success is partly due to one of its main features, which is its small memory usage as shown in Table 4. As shown in this table, the memory usage of FMT-max is very often less than a quarter of the one required by KM17. This allows to FMT to compute the max-truss of much bigger graphs as discussed next.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

NET	MAX-TRUSS	DECOMPOSITION				
	FMT-max	FMT-dec	SL+17	KM17	D18	WG+18
flickrEdges	0.55	**0.55**	6.25	2.35	1021.64	76.47
Amazon0505	1.27	1.27	1.21	**0.35**	2.45	9.43
dblp-2011	1.7	1.7	1.65	**0.5**	4.99	11.86
as-Skitter	6.66	6.66	**5.93**	6.52	320.36	99.19
cit-Patents	7.61	7.61	6.75	**3.79**	7.05	41.59
hollywood-2009	27.71	**58.85**	168.27	90.54	*oot*	4 815
enwiki-2013	50.30	**50.36**	102.36	152.61	*oot*	2 199
hollywood-2011	105.71	**135.61**	357.15	170.29	*oot*	12 576
g500-sc23-ef16	47.13	**154.42**	298.07	690.44	*oot*	*oot*
g500-sc25-ef16	292.6	**775.76**	1 558.06	4 501.8	*oot*	*oot*
arabic-2005	140.69	**349.43**	*oom*	1 445.39	*oot*	*oot*
it-2004	208.34	**607.57**	*oom*	5 784.62	*oot*	*oot*
twitter-2010	505.44	**1 587.33**	*oom*	57 990.07	*oot*	*oot*
gsh-2015-host	1 483.12	**1 957.21**	*oom*	16 978.97	*oot*	*oot*
com-Friendster	402.55	**1 406.47**	*oom*	2 006.91	*oot*	*oot*
gsh-2015	45 449.0†	*oom*	*oom*	*oom*	*oom*	*oom*
wdc2012	545 587.6†	*oom*	*oom*	*oom*	*oom*	*oom*

TABLE 3. Execution time comparison (sec.) using 24 cores. *oot*: required more than 30 000 seconds. †: obtained setting $\mathcal{M} = 3 \cdot 10^9$ and $r = 3$.



(a) as-skitter (b) enwiki-2013 (c) arabic-2005

FIGURE 2. Execution time (sec.) when increasing the numbers of cores.

NET	MAX-TRUSS	DECOMPOSITION		
	FMT-max	FMT-dec	SL+17	KM17
g500-sc23-ef16	1.02	5.95	22.42	4.59
g500-sc25-ef16	4.1	24.01	74.72	18.39
arabic-2005	4.39	25.56	*oom*	19.71
it-2004	8.13	47.37	*oom*	36.41
twitter-2010	9.43	55.22	*oom*	43.09
gsh	11.97	69.55	*oom*	55.52
com-Friendster	14.86	85.02	*oom*	74.66
gsh-2015	121.8†	*oom*	*oom*	*oom*
wdc2012	121.7†	*oom*	*oom*	*oom*

TABLE 4. Memory usage (GiB) comparison using 24 cores. *oom*: out of memory. *oot*: required more than 30 000 seconds. †: obtained setting $\mathcal{M} = 3 \cdot 10^9$ and $r = 3$.

Massive graph experiments

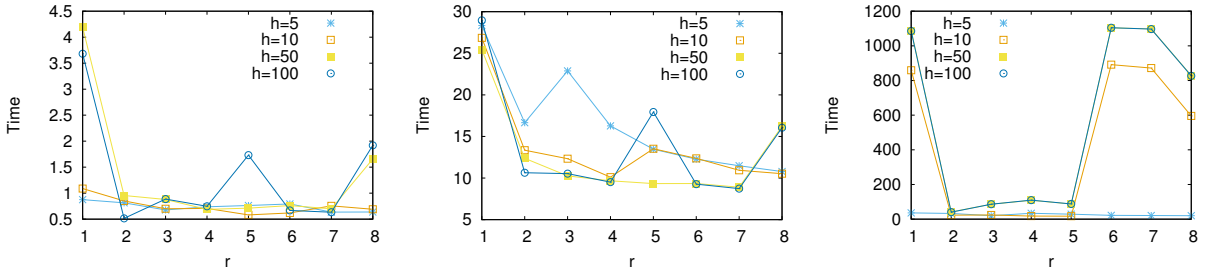
Using FMT, we have computed the max-truss of gsh-2015 and eu-2015, setting $\mathcal{M} = 3 \cdot 10^9$ and $r = 3$. Since the memory available on our computing platform, namely 128G, is not sufficient for all the methods based on truss decomposition, they run *oom*. In the last rows of Table 3 and of Table 4 we report respectively the time and space required by FMT to conclude the experiment.

Scalability

In the following, we discuss the execution time of all the algorithms varying the number of cores. We set the number of cores as 1, 3, 6, 12, and 24. We report the results in Figure 2 for as-skitter, enwiki-2003, and arabic-2005 (both axis are in log scale). Not all the methods are appearing in all the plots since some of them ran *oot* or *oom*.

In the case of Figure 2(a), due to the size of the graph, the differences between the methods are smaller, as also their scaling factors. When increasing the size of the graph, as in the case of (b), the differences become more visible: FMT-dec always outperforms the competitors, namely SL+17 and KM17, which are the only ones able to deal with enwiki-2013. The improvement of FMT-max to compute the max-truss is more than one order of magnitude with respect to computing the truss decomposition using SL+17 and KM17, and it is consistently increasing with the number of cores. This trend is replicated at a higher scale in (c), where SL+17 is not present as it goes *oom* and KM17 spends more than ten times the time required by FMT-dec and FMT-max, confirming the results of Table 3.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access



(a) dblp-2011 (b) enwiki-2013 (c) hollywood-2009

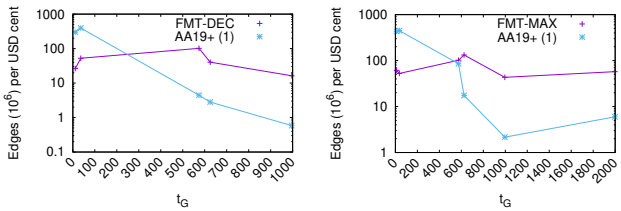
FIGURE 3. Execution time (sec.) of FMT as a function of r, setting $\mathcal{M} = m/h$ for different values of h.

ALGORITHM	HARDWARE	COST/HOUR
FMT	n1-custom-24-131072, SSD 3x375 GB	1.49 USD
PS18	n1-custom-24-131072, SSD 2x375 GB†	1.45 USD
MD+18 M. (1)	n1-custom-20-524288-extended, SSD 4x375 GB 1 x NVIDIA TESLA P100	6.51 USD
MD+18 N. (1)	n1-custom-20-524288-extended, SSD 4x375 GB 1 x NVIDIA TESLA V100	7.53 USD
MD+18 N. (4)	n1-custom-20-524288-extended, SSD 4x375 GB 4 x NVIDIA TESLA V100	14.97 USD
AA+19 (1)	n1-custom-20-524288-extended, SSD 4x375 GB 1 x NVIDIA TESLA V100	7.53 USD
AA+19 (4)	n1-custom-20-524288-extended, SSD 4x375 GB 4 x NVIDIA TESLA V100	14.97 USD

TABLE 5. Parameters for the GCE pricing calculator, single instance with "regular VM class, custom machine type" (updated: Feb 02, 2020). The cost is the hourly rate in US dollars for a single instance of the virtual machine. The numbers (1) and (4) in brackets refer to the number of GPUs used on the instance. Letters M. and N. for MD+18 refer to the architecture used in the experiments (M.: Minksky, N.: Newell). †: PS18 uses either 128 or 256 instances simultaneously in [25].

NETWORK	MAX-TRUSS		DECOMPOSITION						
	FMT-max	AA+19 (1)	FMT-dec	PS18	MD+18 M. (1)	MD+18 N. (1)	MD+18 N. (4)	AA+19 (1)	AA+19 (4)
flickrEdges	**101.78**	85.20	**101.78**	-	0.38	1.53	0.07	4.44	5.59
Amazon0505	61.39	**430.35**	26.31	-	7.88	64.79	-	**299.04**	-
cit-Patents	52.44	**448.72**	52.44	-	10.26	99.96	8.07	394.87	**1986.24**
g500-sc23-ef16	**132.52**	17.45	**40.44**	-	-	0.89	-	2.80	2.35
g500-sc25-ef16	**43.22**	2.15	**16.30**	-	-	-	-	0.58	0.65
twitter-2010	**57.48**	5.98	**18.30**	0.03†	-	-	-	< 0.8	-

TABLE 6. Performance (millions of edges processed per USD cent, higher is better) of FMT-max and AA+19 (for max-truss computation) and FMT-dec, PS18, MD+18, AA+19 (for truss decomposition). The numbers (1) (4) refer to the number of GPUs used on the instance. †: 128 machines employed. To compute this cost we relied on the running times specified in the corresponding papers, and on the cost reported in Table 5.



(a) Truss Decomposition (b) Max Truss

FIGURE 4. Millions of edges processed per USD cent by FMT and AA+19 (1), as a function of t_G on the graphs considered in Table 5.

B. INDIRECT COMPARISON

Two relevant results on truss decomposition are from the IEEE MIT HPEC GraphChallenge 2018 [27]: the champion [25] is based on a high-performance distributed algorithm (hereafter called PS18) and one of the finalists [22] is based on a high-performance collaborative (GPU+CPU) algorithm (hereafter called MD+18). Another relevant result is a winner of the Student Innovation Awards in the GraphChallenge of the following year [2], which also presents a GPU-based approach for truss decomposition and computing the max truss (hereafter called AA+19).

The comparison of PS18, MD+18 and AA+19 with our FMT cannot be directly performed as the computing platforms are pairwise different (furthermore, the software is not available). Indeed, PS18 runs on a cluster of 256 machines, and MD+18 and AA+19 use one or four Nvidia GPUs.

To overcome these differences and to make a uniform

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3011667, IEEE Access

comparison, we simulated the economical cost of running all the algorithms on the Google Compute Engine (GCE), by analyzing the number of processed edges (in millions) that a US dollar cent can buy, using the platforms in GCE. To compute this cost we relied on the running times specified in the corresponding papers [2], [22], [25], and on the cost reported by the GCE pricing calculator[10] for each computing platform. We considered the cost of the closest *more* powerful infrastructure for our results, and that of the closest *less* powerful infrastructures for PS18, MD+18 and AA+19, to do not give an advantage to our algorithm. Regarding the datasets, we selected the graphs which were used by at least two algorithms among PS18, MD+18 and AA+19. We illustrate the hourly costs in Table 5, and the millions of edges processed per USD cent in Table 6.

As it can be observed, the performance of FMT is orders of magnitude higher than the other considered algorithms, except for `Amazon0505` and `cit-Patents`, where AA+19 excels. We observe that the latter cases correspond to graphs with the lowest trussness values: `Amazon0505` has trussness 11 and `cit-Patents` 36, while all the other networks have trussness values over 500. For the sake of completeness, we plotted in Figure 4 the performance per USD against the value of t_G in the considered datasets: we can see that the performance per cost of FMT-dec and FMT-max remains consistent on all considered datasets, while the performance per cost of AA+19 gets significantly worse as t_G increases. The rationale behind this behaviour is that approaches based on matrix multiplication (such a GPU-based ones) re-compute the support of all edges in the graph simultaneously, exploiting the mass-parallelism given by the GPUs, while our combinatorial approach only processes information relative to the edges that are iteratively removed. Moreover, our CPU-based algorithm relies on hardware that is up to one order of magnitude cheaper than the hardware required by other existing algorithms (Table 5), leading to a more cost-effective computation. It follows that approaches such as AA+19 obtain good performance per cost on graphs with only small k-trusses (as fewer multiplications are performed), their performance degrades when denser k-trusses start to appear; on the other hand, the performance of FMT-dec and FMT-max remains consistent on all considered datasets, outperforming all the other considered algorithms by orders of magnitude as t_G increases.

It is worth remarking that PS18 notably concluded an experiment for a further graph called `wdc2012`, having 3.5×10^9 nodes and 128×10^9 edges. The performance of PS18 on this graph is 0.09 million edges per USD cent using 256 machines like the ones described in Table 5 (row PS18). Unfortunately, we were not able to conclude an experiment on this graph on our single machine, as it would require 8x375 GB SSD (instead of our 3x375 GB). However, estimating our performance for `wdc2012`, FMT processes several orders of magnitude more edges per USD cent. This estimation is obtained using the results shown in Table 3, as we concluded an experiment on a slightly smaller graph (smaller enough to use our resources), which is `eu-2015`, using FMT-max. In this case, the performance was 3.57 million edges per USD cent. Scaling the running time (in Table 3) for the size of `wdc2012` and considering the cost of a suitable machine above (1.69 USD/h instead of 1.49 USD/h for bringing disk space from 3x375 GB SSD to 8x375 GB SSD), we can roughly estimate our performance for `wdc2012` as 3.13 million edges per USD cent.

[10]https://cloud.google.com/compute/docs/cpu-platforms

C. CHOOSING THE PARAMETERS

In the main part of the paper, we have defined FMT-max as FMT setting $\mathcal{M} = m/10$ and $r = 4$. In this section, we show the execution time of FMT varying $\mathcal{M}$ and r. In particular, we set $\mathcal{M} = m/h$ varying $h \in \{5, 10, 50, 100\}$ and r in $\{1, 2, 3, 4, 5, 6, 7, 8\}$. We show the results in Figure 3, where for each h we report the time of FMT as a function of r for the graphs `dblp-2011`, `eniwiki-2013`, `hollywood-2009`. It is worth observing that the time series is quite stable for $2 \leq r \leq 5$ for all the values of h. For $r \geq 5$, the time can increase as shown in (c), as the first phase of pruning, whose aim is to select the most promising part of the network to find the max-truss, tends to prune too much and induces the algorithm in performing more iterations, i.e. restarts. Among all the possible choices, we have chosen $\mathcal{M} = m/10$ and $r = 4$, but we note that also many other combinations can get very similar results, meaning that FMT is also quite robust.

VII. CONCLUSIONS AND FUTURE WORKS

In this paper, we have presented a new algorithm for computing k-trusses. We have verified that k-trusses are a useful tool for community detection purposes depending on the application: they can give useful insights about communities when combined with tools of clique detection (like in the case of collaboration networks as `hollywood-2011` and `dblp-2011`), as for instance, it allows to quickly compute maximum cliques, and sometimes yields useful information even when used alone (like for `gsh-2015-host` and `enwiki-2013`). We experimentally showed that our algorithm outperforms the most recent state of the art algorithms on different large networks by up to order of magnitudes. For future work, we plan on further exploring the well-known link between the presence of (quasi)cliques and k-trusses, in order to speed up computationally hard problems such as clique and quasi clique detection.

REFERENCES

[1] N.-W. C. S. A. Edge-disjoint spanning trees of finite graphs. Journal of the London Mathematical Society, s1-36(1):445–450, 1961.

[2] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu. Update on k-truss decomposition on gpu. In 2019 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–7. IEEE, 2019.

[3] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. Algorithmica, 17(3):209–223, 1997.

[4] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In Advances in neural information processing systems, pages 41–50, 2006.

[5] M. Bisson and M. Fatica. Static graph challenge on GPU. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–8, Sept 2017.

[6] P. Boldi, A. Marino, M. Santini, and S. Vigna. Bubing: Massive crawling for the masses. TWEB, 12(2):12:1–12:26, 2018.

[7] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In 13th International Conference on World Wide Web, WWW '04, pages 595–602, 2004.

[8] P.-L. Chen, C.-K. Chou, and M.-S. Chen. Distributed algorithms for k-truss decomposition. In Big Data (Big Data), 2014 IEEE International Conference on, pages 471–480. IEEE, 2014.

[9] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. SIAM Journal on Computing, 14(1):210–223, 1985.

[10] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. National Security Agency Technical Report, 16, 2008.

[11] A. Conte, D. De Sensi, R. Grossi, A. Marino, and L. Versari. Discovering k-trusses in large-scale networks. In 2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018, pages 1–6, 2018.

[12] A. Conte, R. D. Virgilio, A. Maccioni, M. Patrignani, and R. Torlone. Finding all maximal cliques in very large social networks. In 9th International Conference on Extending Database Technology, EDBT 2016., pages 173–184, 2016.

[13] T. A. Davis. Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. In 2018 IEEE High Performance extreme Computing Conference (HPEC), pages 1–6. IEEE, 2018.

[14] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. Proc. VLDB Endow., 9(12):1233–1244, Aug. 2016.

[15] M. Farach-Colton and M. Tsai. Computing the degeneracy of large graphs. In LATIN 2014: Theoretical Informatics - 11th Latin American Symposium, Montevideo, Uruguay, March 31 - April 4, 2014., pages 250–260, 2014.

[16] C. Giatsidis, F. D. Malliaros, D. M. Thilikos, and M. Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. In AAAI, volume 14, pages 44–50, 2014.

[17] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, et al. Quickly finding a truss in a haystack. In High Performance Extreme Computing Conference (HPEC), 2017 IEEE, pages 1–7. IEEE, 2017.

[18] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In 2014 ACM SIGMOD international conference on Management of data, pages 1311–1322. ACM, 2014.

[19] H. Kabir and K. Madduri. Parallel k-truss decomposition on multicore systems. In High Performance Extreme Computing Conference (HPEC), 2017 IEEE, pages 1–7. IEEE, 2017.

[20] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. Proc. VLDB Endow., 8(5):509–520, Jan. 2015.

[21] T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan. Linear algebraic formulation of edge-centric k-truss algorithms with adjacency matrices. In 2018 IEEE High Performance extreme Computing Conference (HPEC), pages 1–7. IEEE, 2018.

[22] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W. Hwu. Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition. In 2018 IEEE High Performance Extreme Computing Conference, HPEC, pages 1–7, 2018.

[23] MIT/Amazon/IEEE. GraphChallenge.org: Raising the bar on graph analytic performance. https://graphchallenge.mit.edu/, 2017. [Online; accessed 22/05/2018].

[24] R. Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–4, Sept 2017.

[25] R. Pearce and G. Sanders. K-truss decomposition for scale-free graphs at scale in distributed memory. In 2018 IEEE High Performance extreme Computing Conference (HPEC), pages 1–6. IEEE, 2018.

[26] M.-E. G. Rossi, F. D. Malliaros, and M. Vazirgiannis. Spread it good, spread it fast: Identification of influential nodes in social networks. In 24th International Conference on World Wide Web, pages 101–102. ACM, 2015.

[27] S. Samsi, V. Gadepally, M. B. Hurley, M. Jones, E. K. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner. Graphchallenge.org: Raising the bar on graph analytic performance. In 2018 IEEE High Performance Extreme Computing Conference, HPEC, pages 1–7, 2018.

[28] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis. Truss decomposition on shared-memory parallel systems. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–6, Sept 2017.

[29] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. Journal of Global Optimization, 37(1):95–111, Jan 2007.

[30] C. Voegele, Y. S. Lu, S. Pai, and K. Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–7, Sept 2017.

[31] J. Wang and J. Cheng. Truss decomposition in massive networks. VLDB Endowment, 5(9):812–823, 2012.

[32] Y. Wang, S. Cai, and M. Yin. Two efficient local search algorithms for maximum weight clique problem. In AAAI, pages 805–811, 2016.

[33] D. Wen, L. Qin, X. Lin, Y. Zhang, and L. Chang. Enumerating k-vertex connected components in large graphs. arXiv preprint arXiv:1703.08668, 2017.

[34] V. V. Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on, pages 645–654. IEEE, 2010.

[35] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–7, Sept 2017.

[36] J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo. K-truss decomposition of large networks on a single consumer-grade machine. In 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), volume 00, pages 873–880, August 2018.